Testar operações sobre colecções

- Operações que modificam a colecção
- Operações que visitam a colecção para recolher (acumular, filtrar, ...) informação
- Operações que comparam colecções

Operações que modificam a colecção

- Inserções (rever addMusic())
- Remoções (rever deleteMusic())
- ~~Reordenações~~

Operações que visitam a colecção para recolher informação

Teste à operação `numMusics()`


```
@Test
public void testNumMusics() {
  assertEquals(0, play.numMusics("Marco Paulo"));
  assertEquals(0, play.numMusics("Lady Gaga"));
  assertEquals(0, play.numMusics("Metallica"));
  play.addMusic(m1); // Ser benfiquista, Picarra
  play.addMusic(m2); // Eu tenho 2 amores, Marco Paulo
  play.addMusic(m3); // Master of Puppets, Metallica
  assertEquals(1, play.numMusics("Marco Paulo"));
  assertEquals(0, play.numMusics("Lady Gaga"));
  assertEquals(1, play.numMusics("Metallica"));
  play.addMusic(new MusicClass("Maravilhoso Coracao",
                "Marco Paulo", FileFormat.MP3, 190, 3));
  assertEquals(2, play.numMusics("Marco Paulo"));
  play.addMusic(new MusicClass("Morena oh morenita",
                "Marco Paulo", FileFormat.WAV, 200, 4));
  assertEquals(3, play.numMusics("Marco Paulo"));
  play.deleteMusic(m2); // Eu tenho 2 amores
  assertEquals(2, play.numMusics("Marco Paulo"));
  assertEquals(0, play.numMusics("Lady Gaga"));
  assertEquals(1, play.numMusics("Metallica"));
}
```

- O teste falha, porque o método numMusics ainda tem a implementação por omissão

```
@Override
public int
numMusics(String artist){
  return 0;
}
```

Implementação de `numMusics()`


```
public int numMusics(String artist) {
  int sum = 0;
  for (Music m: myList.values())
    if (m.getArtist().equals(artist))
      sum += 1;
  return sum;
}
```

Teste à operação `numArtists()`


```
@Test
public void testNumArtists() {
  assertEquals(0, play.numArtists());
  assertEquals(0, play2.numArtists());
  play.addMusic(m1); // Ser benfiquista, Picarra
  play.addMusic(m2); // Eu tenho 2 amores, Marco Paulo
  play.addMusic(m3); // Master of Puppets, Metallica
  assertEquals(3, play.numArtists());
  play.addMusic(new MusicClass("Maravilhoso Coracao",
                "Marco Paulo", FileFormat.MP3, 190, 3));
  assertEquals(3, play.numArtists());
  play.addMusic(new MusicClass("Morena oh morenita",
                "Marco Paulo", FileFormat.WAV, 200, 4));
  assertEquals(3, play.numArtists());
  play.deleteMusic(m2); // Eu tenho 2 amores
  assertEquals(3, play.numArtists());
  play.deleteMusic(m3); // Master of Puppets, Metallica
  assertEquals(2, play.numArtists());
  assertEquals(0, play2.numArtists());
}
```

- O teste falha, porque o método numArtists ainda tem a implementação por omissão

```
@Override
public int numArtists(){
  return 0;
}
```

Implementação de `numArtists()`


```
public int numArtists() {
  Set<String> artists = new HashSet<String>(getSize());
  for (Music m: myList.values())
       artists.add(m.getArtist());
  return artists.size();
}
```

Criação de músicas factorizada


```
public class MusicTester {
  private Music sm1;
  private Music sm2;
  private Music m1;
  private Music m2;
  private Music m3;
  private Music m4;
  @Before
  public void setup() {
    sm1 = new MusicClass("Not on the test");
    sm2 = new MusicClass("Lumberjack Song");
    m1 = new MusicClass("Ser Benfiquista", "Luis Picarra",
                        FileFormat.AIFF, 184, 2);
    m2 = new MusicClass("Eu tenho dois amores", "Marco Paulo",
                        FileFormat.MP3, 194, 3);
    m3 = new MusicClass("Master of Puppets", "Metallica",
                        FileFormat.WAV, 452, 4);
    m4 = new MusicClass("Master of Puppets", "Metallica",
                        FileFormat.WAV, 452, 4);
  }
  ...
```

Teste à operação `getDuration()`

- Exercício

Implementação de `getDuration()`


```
public int getDuration() {
   int sum = 0;
   for(Music m: myList.values())
      sum += m.getDuration();
   return sum;
}
```

Acrescentar músicas


```
public class PlayListClass implements PlayList {
  private String name;
  private SortedMap<String,Music> myList;

  public PlayListClass(String name) {
    this.name = name;
    this.myList = new TreeMap<String, Music>();
  }
  private String key(Music m) {
    return m.getName() + m.getArtist();
  }
  @Override
  public boolean contains(Music m) {
    return myList.containsKey(key(m));
  }
  @Override
  public void addMusic(Music m) {
    if ((m != null) && (!this.contains(m)))
      myList.put(key(m), m);
  }
}
```

- Temos de criar uma variável de instância para guardar as músicas
- Actualizar o construtor
- Gerar chave apropriada
- Testar se música faz parte da colecção
- Acrescentar música à colecção, se não existir

Teste à implementação de `mostProductiveArtist()`

- Exercício

Implementação de `mostProductiveArtist()`


```
@Override
public String mostProductiveArtist() {
  int bestScore = 0;
  String bestName = "";
  Map<String, Integer> artists = new HashMap<String, Integer>(getSize());
  for (Music m: myList.values()) {
    int value = 1;
    if (artists.containsKey(m.getArtist()))
      value += artists.get(m.getArtist());
    artists.put(m.getArtist(), value);
    if (bestScore < value) {
        bestScore = value;
        bestName = m.getArtist();
    }
  }
  return bestName;
}
```

Testar propriedades de colecções

- Verificar se uma colecção guarda os elementos da forma esperada

Testar se uma colecção está ordenada

- Exercício:
  - Escolha uma das colecções usadas num dos programas vistos nas aulas teóricas ou práticas e ordene essa colecção
  - Construa um teste que verifique se a sua ordenação funciona tal e qual o esperado, ou não
    - Sugestão: depois de ordenar a colecção, obtenha um iterador e use-o para visitar a colecção verificando a cada passo se o elemento que está a visitar quebra a ordenação estabelecida; se nenhum a quebrar, a colecção ficou bem ordenada

Melhorar a implementação

A funcionalidade está implementada mas pode ser melhorada. Vamos aproveitar estarmos com a **barra verde** para efectuar algumas melhorias, servindo-nos dos testes (correndo-os frequentemente) para garantir que não são introduzidas **regressões**.

Melhorar a implementação da PlayList


```
public class PlayListClass implements PlayList {
  private String name;
  private SortedMap<String,Music> myList;
  private Map<String,Integer> artists;

  public PlayListClass(String name) {
    ...
    this.artists = new HashMap<String, Integer>();
  }
  ...
  @Override
  public void addMusic(Music m) {
    if ((m != null) && (!this.contains(m)))
      myList.put(key(m), m);
      int value = 1;
      if (artists.contains(m.getArtist()))
        value += artists.get(m.getArtist());
      artists.put(m.getArtist(), value);
  }
}
```

- Criamos um mapa adicional que guarda o número de músicas de cada artista

Implementação de **numMusics** – antes e depois


```
@Override
public int numMusics(String artist) {
  int sum = 0;
  for (Music m: myList.values())
    if (m.getArtist().equals(artist))
      sum += 1;
  return sum;
}
```

```
@Override
public int numMusics(String artist) {
  Integer num = artists.get(artist);
  if (num == null)
     return 0;
  return num;
}
```

Implementação de `numArtists()` - antes e depois


```
@Override
public int numArtists() {
  Set<String> artists = new HashSet<String>(getSize());
  for (Music m: myList.values())
    if (!artists.contains(m.getArtist()))
      artists.add(m.getArtist());
  return artists.size();
}
```

```
@Override
public int numArtists() {
  return artists.size();
}
```

Implementação de `mostProductiveArtist()` - antes e depois


```
@Override
public String mostProductiveArtist() {
  int bestScore = 0;
  String bestName = "";
  Map<String, Integer> artists = new HashMap<String, Integer>(getSize());
  for (Music m: myList.values()) {
    int value = 1;
    if (artists.containsKey(m.getArtist()))
      value += artists.get(m.getArtist());
    artists.put(m.getArtist(), value);
    if (bestScore < value) {
        bestScore = value;
        bestName = m.getArtist();
    }
  }
  return bestName;
}
```

```
@Override
public String mostProductiveArtist() {
  int bestScore = 0;
  String bestName = "";
  for (Entry<String,Integer> e: artists.entrySet())
    if (e.getValue() > bestScore) {
      bestName = e.getKey();
      bestScore = e.getValue();
    }
  return bestName;
}
```

Para concluir sobre os testes...

- Definição
- Processo de teste

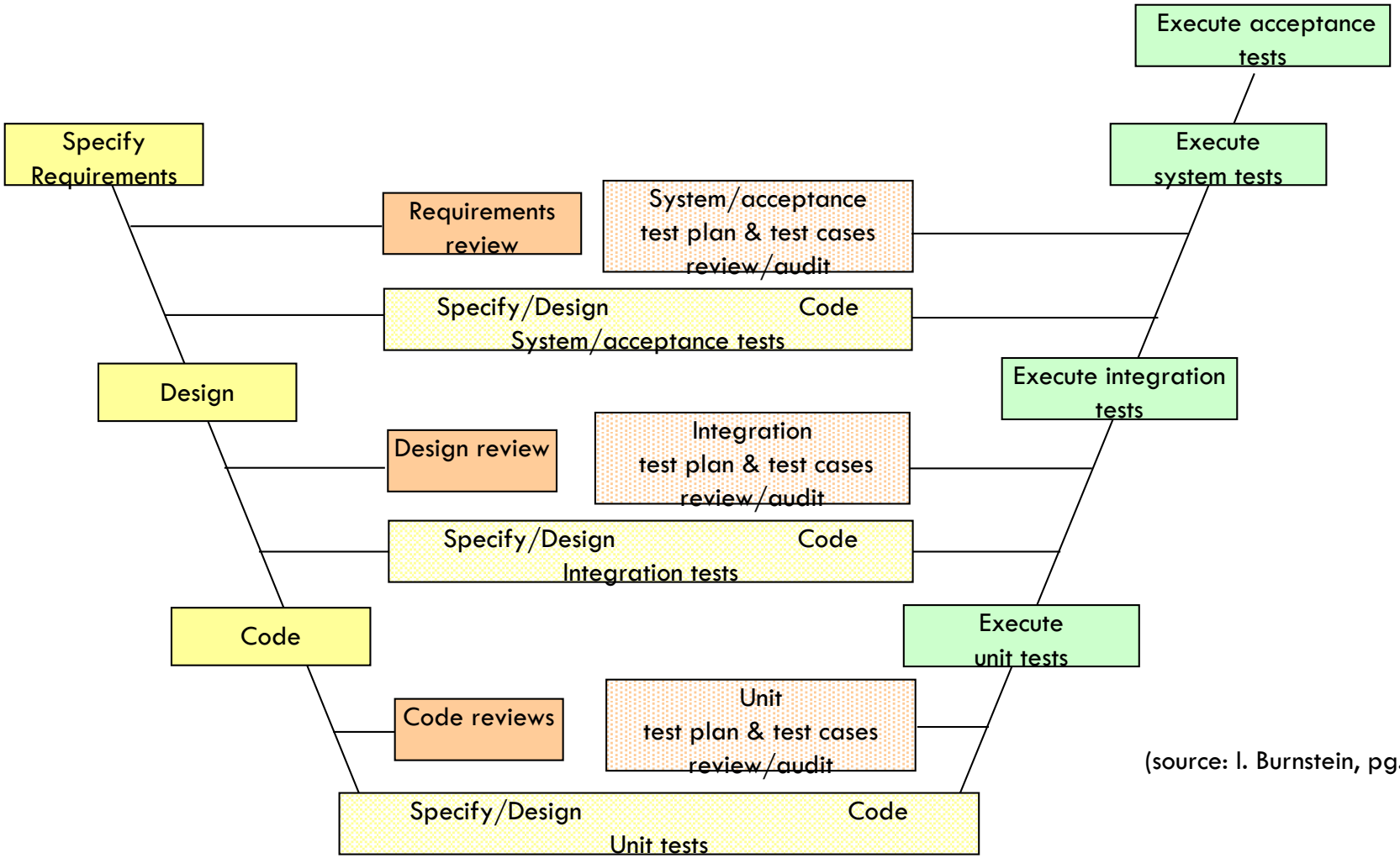
O que é o teste de software?

- O teste de software consiste na verificação **dinâmica** de que o comportamento de um programa, de acordo com um conjunto **finito** de **casos de teste**, escolhidos de modo apropriado de entre um conjunto normalmente infinito de testes possíveis, cumpre o comportamento **esperado**

Implicações da definição de teste

- O teste implica executar o programa com determinados inputs
- Normalmente, é impossível testar todas as situações possíveis, por serem infinitas
- Podemos e devemos usar estratégias de desenho de testes para criar bons casos de teste
- Dado um teste, temos de ser capazes de decidir se um resultado é ou não aceitável
  - Esta decisão é conhecida como o problema do oráculo

Processo de teste (Modelo em V estendido)



(source: I. Burnstein, pg.15)

Fases de teste – Teste Unitário

- Teste de unidades individuais, ou de grupos de unidades relacionadas
- Tipicamente, um tipo de teste sobre a API
- Tipicamente, da responsabilidade do programador
- Testes baseados em experiência, especificações e no código
  - Mas o programador pode seguir estratégias com provas dadas!
- O principal objectivo é detectar defeitos funcionais e estruturais na unidade a testar

Processo de testes – Testes de integração

- Teste em que os componentes são combinados e testados para avaliar a interacção entre eles
- Normalmente, são da responsabilidade de uma equipa de testes independente de quem programou os componentes
- Os testes são baseados numa especificação do sistema
  - Especificações técnicas, desenho do sistema
- O principal objectivo é detectar defeitos que ocorram devido a interacções entre unidades diferentes, e que sejam visíveis ao nível da interface

Fases de teste – Testes de sistema

- Testes realizados sobre um sistema completo, de modo a avaliar se o sistema está de acordo com os requisitos para ele especificados
  - Requisitos funcionais e não-funcionais (ex. Eficiência)
- Testes funcionais de caixa-negra, através da interface com o utilizador, normalmente baseados em informação recolhida no documento de requisitos
  - Estes testes ignoram completamente os detalhes de implementação do sistema
- Normalmente, são testes realizados por uma equipa independente

Fases de teste – Testes de Aceitação

- Testes formais destinados a determinar se um sistema satisfaz, ou não, um determinado conjunto de critérios de aceitação, de modo a que o cliente possa decidir se deve ou não aceitar o sistema
- Testes formais realizados para permitir que o utilizador, cliente, ou outra entidade autorizada aceitam um sistema, ou um componente
  - Tipicamente realizados pelo cliente
  - Frequentemente os testes são baseados num documento de requisitos, ou no manual do utilizador
  - O principal motivo é verificar se os requisitos e expectativas são atingidos

Fases do teste – Testes de Regressão

- Repetição selectiva de testes de um sistema, ou de um componente, para verificar que:
  - As modificações feitas desde o último teste não provocaram efeitos indesejados
  - O sistema, ou componente, continuam a satisfazer os requisitos especificados

Testes

- O mecanismo de testes pode demonstrar a existência de defeitos, mas nunca a sua ausência
- No contexto de POO, mal arranhamos a superfície da prática efectiva de testes de software